Checkpointing Workflows for Fail-Stop Errors

Li Han, Louis-Claude Canon, Henri Casanova, Yves Robert, Frédéric Vivien

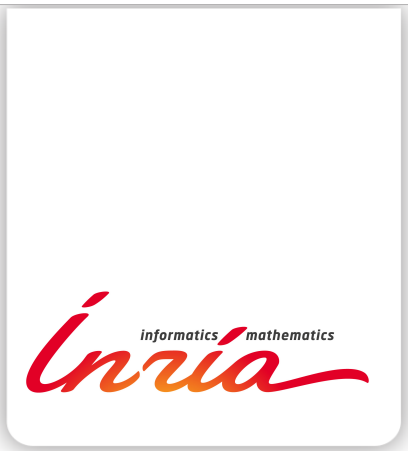
RESEARCH REPORT

N° 9068

November 2017

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9068--FR+ENG



Checkpointing Workflows for Fail-Stop Errors

Li Han*†, Louis-Claude Canon‡, Henri Casanova§, Yves Robert*¶, Frédéric Vivien*

Project-Team ROMA

Research Report n° 9068 — November 2017 — 33 pages

* LIP, École Normale Supérieure de Lyon, CNRS & Inria, France
† East China Normal University, China
‡ FEMTO-ST, Université de Bourgogne Franche-Comté, France
§ University of Hawai'i at Manoa, USA
¶ Univ. Tenn. Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Abstract: We consider the problem of orchestrating the execution of workflow applications structured as Directed Acyclic Graphs (DAGs) on parallel computing platforms that are subject to fail-stop failures. The objective is to minimize expected overall execution time, or makespan. A solution to this problem consists of a schedule of the workflow tasks on the available processors and of a decision of which application data to checkpoint to stable storage, so as to mitigate the impact of processor failures. For general DAGs this problem is hopelessly intractable. In fact, given a solution, computing its expected makespan is still a difficult problem. To address this challenge, we consider a restricted class of graphs, Minimal Series-Parallel Graphs (M-SPGs). It turns out that many real-world workflow applications are naturally structured as M-SPGs. For this class of graphs, we propose a recursive list-scheduling algorithm that exploits the M-SPG structure to assign sub-graphs to individual processors, and uses dynamic programming to decide which tasks in these sub-gaphs should be checkpointed. Furthermore, it is possible to efficiently compute the expected makespan for the solution produced by this algorithm, using a first-order approximation of task weights and existing evaluation algorithms for 2-state probabilistic DAGs. We assess the performance of our algorithm for production workflow configurations, comparing it to (i) an approach in which all application data is checkpointed, which corresponds to the standard way in which most production workflows are executed today; and (ii) an approach in which no application data is checkpointed. Our results demonstrate that our algorithm strikes a good compromise between these two approaches, leading to lower checkpointing overhead than the former and to better resilience to failure than the latter. To the best of our knowledge, this is the first scheduling/checkpointing algorithm for workflow applications with fail-stop failures that considers workflow structures more general than mere linear chains of tasks.

Key-words: workflow, checkpoint, fail-stop error, resilience.

Stratégies de checkpoint pour les workflows en présence d'erreurs fatales

Résumé : Ce rapport considère l'ordonnancement de workflows (applications structurées en forme de graphes de tâches acycliques, ou DAGs) sur des plates-formes parallèles à grande échelle, soumises à des erreurs fatales. L'objectif est de minimiser l'espérance du temps total d'exécution, ou makespan. Une solution à ce problème comprend l'allocation ordonnée des tâches aux processeurs, et les décisions de checkpoint: quelles tâches sont suivies d'un checkpoint? Même pour une solution donnée, le calcul du makespan reste difficile. Nous nous restreignons à une classe de DAGs particuliers, les graphes séries-parallèles minimaux, ou M-SPGs. De nombreux workflows issus des applications ont pour graphe un M-SPG. Pour de tels graphes, nous proposons un algorithme qui utilise la structure récursive du M-SPG pour allouer des sous-graphes à chaque processeur, et utilise la programmation dynamique pour décider quelles tâches checkpointer. Il est alors possible de calculer efficacement le makespan via des algorithmes d'évaluation de DAGs probabilisites à deux états. Nous établissons expérimentalement la performance de notre approche en la comparant, sur des workflows applicatifs bien connus, avec l'approche qui checkpointe toutes les tâches et celle qui n'en checkpointe aucune. Les résultats montrent que notre approche réalise un bon compromis entre les deux approches extrêmes, avec moins de surcoût de checkpoint que la stratégie qui checkpointe tout le temps, et une meilleure résilience que celle qui ne checkpointe jamais. A notre connaissance, notre approche est la première à considérer des DAGs plus généraux que des chaînes pour les erreurs fatales.

Mots-clés : workflow, checkpoint, erreur fatale, résilience.

1 Introduction

This paper proposes a new algorithm to execute workflows on parallel computing platforms subject to fail-stop processor failures, e.g., a large-scale cluster. In the case of a fail-stop error, the execution stops and has to be restarted from scratch, either on the same processor once it reboots or on a spare. The de-facto approach to handle such failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution. Because workflows are structured as Directed Acyclic Graphs (DAGs) of tasks, they are good candidates for a C/R approach. First, tasks can be checkpointed individually and asynchronously. Second, rather than checkpointing the entire memory footprint of a task, it is only necessary to checkpoint its output data.

The common strategy used in practice is *checkpoint everything*, or CkptAll: all output data of each task is saved onto stable storage (in which case we say "the task is checkpointed"). For instance, in production Workflow Management Systems (WMSs) [1, 2, 3, 4, 5, 6], the default behavior is that all output data is saved to files and all input data is read from files, which is exactly the CkptAll strategy. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum would be a *checkpoint nothing* strategy, or CkptNone, by which all output data is kept in memory (up to memory capacity constraints) and no task is checkpointed. This corresponds to "in-situ" workflow executions, which has been proposed to reduce I/O overhead [7]. The downside is that, in case of a failure, a large number of tasks may have to be re-executed, leading to slow restarts. The objective of this work is to achieve a desirable trade-off between these two extremes.

Consider the problem of scheduling a workflow execution on failure-prone processors and deciding how to checkpoint the tasks. The objective is to minimize the expectation of the execution time, or makespan. The makespan is a random variable because task execution times are probabilistic, due to failures causing task re-executions. The complexity of this problem is steep. In fact, the complexity of computing the expected makespan of a given solution is already difficult. A solution consists of an ordered list of tasks to execute for each processor; and for each task whether or not to save its output data to stable storage after its execution. In a failure-free execution, the makespan of a solution is simply the longest path in the DAG, accounting for serialized task executions at each processor. With failures, however, estimating the expected makespan of a solution is difficult. Consider the CkptAll strategy and a solution in which each task is assigned to a different processor. Computing the expected makespan amounts to computing the expected longest path in the schedule. Unfortunately, computing the expected length of the longest path in a DAG with probabilistic task durations is a known difficult problem [8, 9]. Even in the simplified case when task durations are random variables that can take only two discrete values, the problem is #P-complete [8].[1] As a result, several approximation methods have been developed to estimate the expected longest path of a DAG with probabilistic task durations, including Monte-Carlo simulations (see Section 3.4 for a detailed discussion).

The aim of this paper is to design a strategy that achieves better performance than CkptAll and than CkptNone by checkpointing some, but not all, tasks. When all tasks are checkpointed, failures are contained since a task can just be restarted after a failure by reading

[1] Recall that #P is the class of counting problems that correspond to NP decision problems [10, 11, 12], and that #P-complete problems are at least as hard as NP-complete problems.

input data that has been saved to stable storage by that task's checkpointed predecessors. This is no longer the case when some tasks are not checkpointed, which gives rise to new difficulties when trying to estimate the expected makespan (and thus minimize it). This is because a failure on a processor can lead to the re-execution of tasks on other processors. Furthermore, identifying which tasks must be re-executed after a checkpoint depends on how inter-processor communication is performed (e.g., at which time a task's output data is sent to another processor after that task completes, whether the sent data is kept in the memory of the sender and for how long). Section 2 provides a detailed example that highlights these difficulties. For simplicity we refer to these difficulties as "processor interference" because a failure on a processor can cause task re-executions on other processors.

Our key observation is that a way of avoiding processor interference completely, and thus of designing of a strategy that checkpoints only some tasks, is to prohibit "crossover dependencies". We define a crossover dependency as a dependency between a task T and a direct successor T' that are scheduled on different processors, where the output data of T is not checkpointed. Prohibiting crossover dependencies reduces the difficulty of our problem in four ways. 1) As discussed above and in Section 2, with crossover dependencies a few failures can lead to many task re-executions and data re-transfers, during which other failures can occur. Prohibiting crossover dependencies avoids such complex scenarios because failures are contained to a single processor, thus enabling simple task restarts. 2) Without crossover dependencies, there is no longer any need for inter-processor communications and thus for any assumptions regarding these communications. The local storage/memory at each processor is limited to storing data that is input to tasks that will execute on that same processor. 3) Without crossover dependencies, it is possible to determine the optimal set of tasks to checkpoint for groups of tasks assigned to a single processor. 4) Without crossover dependencies, computing the expected makespan reduces, as for CkptAll, to computing the longest path in a DAG with probabilistic task durations.

Our approach is to avoid crossover dependencies by restricting the problem to a particular class of workflow DAGs: Minimal Series Parallel Graphs (M-SPGs) [13]. Despite their name, M-SPGs extend classical Series Parallel Graphs (SPGs) [14] by allowing source and sink nodes to not be merged in series composition (see Section 3.1 for details). It turns out that most real-world workflows, e.g., those executed today by production WMSs [1, 2, 3, 4, 5, 6], are M-SPGs. The structure of these graphs makes it possible to orchestrate the execution in fork-join fashion, by which processors compute independent sets of tasks, before joining and exchanging data with other processors. We call these independent sets of tasks *superchains*, because tasks in these sets are linearized into a chain (because they are executed by a single processor) but have forward dependencies that can "skip over" immediate successors. We remove all crossover dependencies by always checkpointing the output data of the exit tasks of a superchain, thus removing the difficulties associated with processor interference.

In this work we propose, to the best of our knowledge, the first scheduling/checkpointing strategy for minimizing the expected makespan of workflow applications with fail-stop failures that considers workflow structures more general than mere linear chains of tasks. More specifically, our contributions are:

- A scheduling/checkpointing strategy, CkptSome, for M-SPGs that improves upon both the de-facto standard CkptAll strategy and the CkptNone strategy (Section 3). CkptSome avoids all crossover dependencies and relies on the two algorithms below;
- A list-scheduling algorithm, which is inspired by the "proportional mapping" approach [15], for scheduling M-SPG workflows as sets of superchains (Section 4);

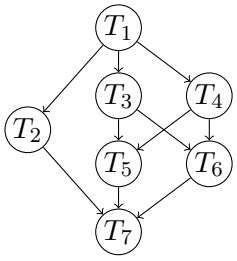


Figure 1: Example task graph.

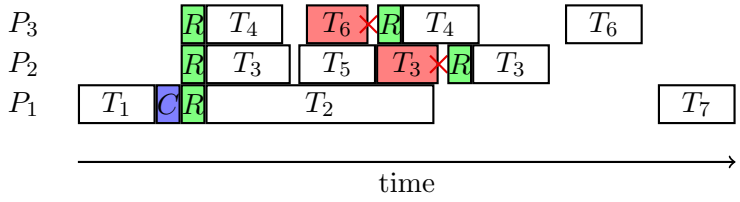


Figure 2: Example schedule of the workflow in Figure 1 on 3 processors. Processor P_3 fails while executing task T_6 and has to start all its tasks again. However, processor P_2 then fails and the result of task T_3 is lost, delaying the re-execution of task T_6 until task T_3 is re-executed on processor P_2.

• An algorithm, which extends the dynamic programming algorithm by Toueg and Babaoğlu [16], to checkpoint tasks in a superchain optimally (Section 5);
• The #P-completeness of the problem of computing the expected makespan for the CkptNone strategy (Section 6). To the best of our knowledge, the complexity of computing, or even approximating, the expected makespan for CkptNone was an open problem.
• Experimental evaluation with real-world Pegasus [1] workflows to quantify the performance gains afforded by our proposed approach in practice (Section 7).
In addition to the above sections, Section 2 details an example with crossover dependencies, Section 8 reviews relevant related work, and Section 9 provides concluding remarks and highlights directions for future work.

2 Example

Consider the workflow in Figure 1, which comprises 7 tasks, T_i, $1 \leq i \leq 7$. The execution of this workflow on 3 processors, P_i, $1 \leq i \leq 3$, for a given schedule is shown in Figure 2. Two failures occur during the execution, first on P_3 and then on P_2 (shown as red X's on the figure). In this example, T_1 is checkpointed. The checkpoint overhead for saving T_1's output to stable storage occurs immediately after T_1 completes (shown as C on the figure). All successors of T_1 must then "recover" from that checkpoint to begin execution, which also has some overhead (shown as R's on the figure before the execution of T_2, T_3 and T_4). No other task besides T_1 is checkpointed in this example. As a result, some direct (i.e., not via stable storage) communication is required between some tasks. Figure 2 shows delays due to these communications. For instance, the first execution of T_6 on P_3 does not start immediately after T_3 completes but only after a delay, which corresponds to the time for P_2 to send T_3's output to P_3.

For non-checkpointed tasks and their successors, one must define precisely how inter-processor communications take place, i.e., when the data is transferred and for how long it is stored at the sender and the receiver. Recall that when a failure occurs on a processor, the whole content of that processor's memory is lost. As a result, the way in which inter-processor communications take place can impact the failure recovery procedure. For instance, in Figure 2, a failure strikes P_2 after the completion of T_5 but before T_7 begins executing on P_1. If the output data of T_5 is sent to P_1 as soon as T_5 completes, there is no need to re-execute T_5 for executing T_7. On the contrary, if this output is sent as late as possible (i.e., so that it is received just before the execution of T_7 begins), then T_5 will need to be

re-executed because its output will have been lost on P_2 due to the failure. For the sake of the example in Figure 2, we have chosen the former option. More formally, consider a non-checkpointed task T, executed on a processor P, that produces data that is needed by task T' on a different processor P' (i.e., a crossover dependency as described in Section 1). Then, the data is transferred from P to P' immediately after T completes. This data is deleted from memory on P as soon as the data transfer has completed, and deleted in memory on P' after T' completes. If T and T' are scheduled on the same processor P, then the output data of T is in memory of P from the completion of T until the completion of T'.

There are three phases in the schedule in Figure 2: (i) execution until the first failure; (ii) recovery and execution until the second failure; and (iii) recovery and termination of the execution. The result of the execution of T_1 on processor P_1 is saved to stable storage because T_1 is checkpointed. Therefore, T_1 will never need to be re-executed once it has executed successfully. The three successors of T_1, namely T_2, T_3 and T_4, start their executions after reading the output data of T_1 from stable storage. Upon completion, the results of T_3 and T_4 are transmitted to their successors on other processors immediately.

The first failure interrupts the execution of T_6 on processor P_3. Due to this failure, P_3 loses the output data of T_3 and T_4, which are required to execute T_6. Hence, both T_3 and T_4 must be restarted on processors P_2 and P_3, respectively. The result of T_1 is recovered from stable storage in order to allow the execution of T_4 on P_3. Processor P_2 still has the output of T_1 in memory, and thus does not perform this recovery. However, at the time of the failure, T_5 is running on P_2, and then the re-execution of T_3 can only start once T_5 completes. This is because we have made the common implicit assumption that tasks are non-preemptible.

The second failure interrupts the re-execution of task T_3 on P_2. P_2 then re-executes T_3, which requires recovering the output of T_1 from stable storage. Once T_3 completes on P_2, then P_3 can execute T_6. Finally, T_7 is executed on processor P_1 after T_6 completes.

This example highlights the difficulties caused by crossover dependencies, here from T_3 to T_6 (where a failure on P_3 causes a re-execution on P_2) and from task T_5 to task T_7 (for which a failure may cause a re-execution depending on assumptions on when inter-processor communications are performed). The main observation is that, with crossover dependencies, a failure on a processor can cause task re-executions on other processors. These re-executions are themselves subject to failures, and these failures can also cause re-executions on yet other processors. As a result, a failure on one processor can "ripple" through all processors. As a result, estimating (and thus minimizing) the expected makespan is hopelessly combinatorial. None of the known methods designed to approximate the expected makespan of DAGs with probabilistic task durations can be applied. To the best of our knowledge, the only option would be to use discrete event simulation and hope to estimate the expected makespan as an average over a (prohibitively?) large number of trials with randomly injected failures.

In the example in Figure 2, avoiding all crossover dependencies would require checkpointing four additional tasks: T_3, T_4, T_5 and T_6. With these additional checkpoints, a task re-execution on a processor only happens if a failure occurs on that processor. This avoids the failure rippling effect described above. Without crossover dependencies, we are back to a situation where failures are contained to individual processors, just as when all tasks are checkpointed in the CkptAll approach.

Given the above, in this work we avoid crossover dependencies altogether. Once these dependencies are eliminated, our approach views all the tasks executed by a same processor in between two checkpoints as a single (larger) task. In other terms, we logically coalesce a group of consecutive not-checkpointed tasks followed by a checkpointed task into a single

checkpointed task. With this logical coalescing, we can evaluate the expected makespan as that of a DAG with probabilistic task durations, for which approximation methods exist. In summary, prohibiting crossover dependencies not only contains the impact of failures, but it also enables us to estimate the expected makespan of a solution.

3 Preliminaries and Proposed Approach

In this section, we define M-SPGs, the class of workflow DAGs that we consider in this work. We then detail the fault-tolerance model for failures and checkpoints. Next, we briefly review methods to compute the expected longest path in a DAG with probabilistic task durations. Finally, we provide an overview of our proposed approach, including how we schedule and checkpoint tasks.

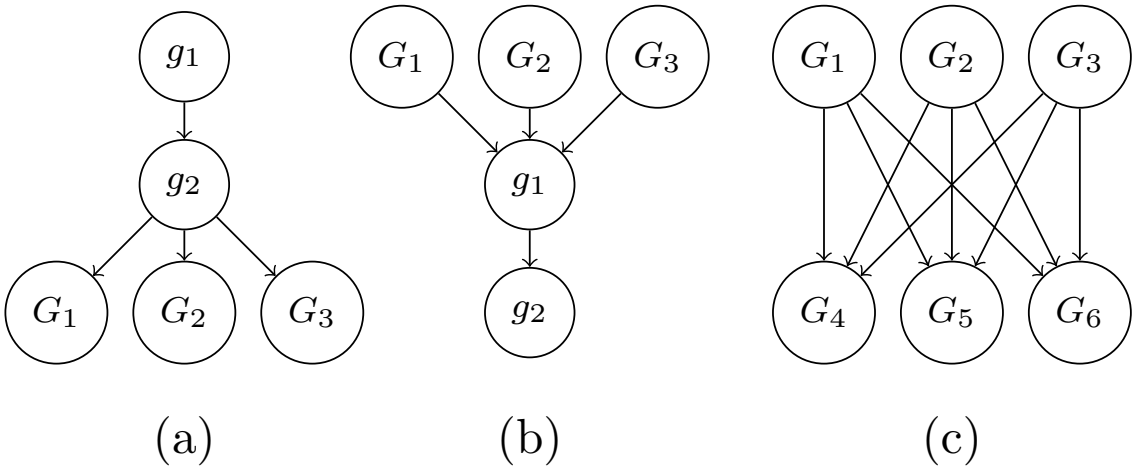


Figure 3: Example M-SPG structures (g_1 and g_2 are atomic tasks whereas G_1 to G_6 are M-SPGs): (a) fork: $(g_1 \overrightarrow{;} g_2) \overrightarrow{;} (G_1||G_2||G_3)$; (b) join: $(G_1||G_2||G_3) \overrightarrow{;} (g_1 \overrightarrow{;} g_2)$; (c) bipartite: $(G_1||G_2||G_3) \overrightarrow{;} (G_4||G_5||G_6)$.

3.1 Minimal Series Parallel Graphs (M-SPG)

We consider computational workflows structured as Minimal Series Parallel Graphs (M-SPGs) [13], which (despite their name) are generalizations of standards SPGs [14]. An M-SPG is a graph $G = (V, E)$, where V is a set of vertices (representing workflow tasks) and E is a set of edges (representing task dependencies). Each task has a *weight*, i.e., its execution time in a failure-free execution. An M-SPG is defined recursively based on two operators, $\overrightarrow{;}$ and $||$, defined as follows:

• The *serial composition* operator, $\overrightarrow{;}$, takes two graphs as input and adds dependencies from all sinks of the first graph to all sources of the second graph. Formally, given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1 \overrightarrow{;} G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup (sk_1 \times sc_2))$, where sk_1 is the set of sinks of G_1 and sc_2 the set of sources of G_2. This is similar to the serial composition of SPGs, but without merging the sink of the first graph to the source of the second, and extending the construct to multiple sources and sinks.

• The *parallel composition* operator, $||$, simply makes the union of two graphs. Formally,

given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $G_1||G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. This is similar to the parallel composition of SPGs, but without merging sources and sinks.

Given the above operators, an M-SPG is then defined recursively as follows:

- A *chain* $g_1 \overrightarrow{;} \ldots \overrightarrow{;} g_n$, where each g_i is an atomic task;
- A serial composition $G_1 \overrightarrow{;} \ldots \overrightarrow{;} G_n$, where each G_i is an M-SPG; or
- A parallel composition $G_1|| \ldots ||G_n$, where each G_i is an M-SPG.

Figure 3 shows example M-SPG structures. Due to the above definition supporting multiple sources and sinks, and not merging sources and sinks, M-SPGs naturally support fork, join (and therefore fork-join), and bipartite structures. It turns out that these structures are common in production workflow applications. For instance, most workflows from the Pegasus benchmark suite [17, 1] are M-SPGs. Overall, M-SPGs exhibit the recursive structure of SPGs (which is key to developing tractable scheduling/checkpointing solutions), but are more general, and as a result maps directly to most production workflow applications. In particular, M-SPGs can model communication patterns that cannot be modeled with SPGs (this is the case of the bipartite structure shown in Figure 3(c)).

3.2 Fault-tolerance model

In this work, we consider failure-prone processors that stop their execution once a failure occurs (i.e., we have fail-stop errors). Computation has to be started from scratch, either on the same processor after a reboot, or on a spare processor.

Consider a single task T, with weight w, scheduled on such a processor, and whose input is stored on stable storage. It takes a time r to read that input data from stable storage, either for its first execution or after a failure. The total execution time W of T is a random variable, because several execution attempts may be needed before the task succeeds.

Let $\lambda \ll 1$ be the Exponential failure rate of the processor. With probability $e^{-\lambda(r+w)}$, no failure occurs, and W is equal to $r+w$. With probability $(1-e^{-\lambda(r+w)})$, a failure occurs. For Exponentially distributed failures, the expected time to failure, knowing that a failure occurs during the task execution (i.e., in the next $r+w$ seconds), is $1/\lambda - (r+w)/(e^{\lambda(r+w)}-1)$ [18]. After this failure, there is a downtime d, which is the time to reboot the processor or migrate to a spare. Then we start the execution again, first with the recovery r and then the work w. With a general model where an unbounded number of failures can occur during recovery and work, the expected time W to execute task T is given by $W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w)} - 1\right)$ [18]. Now if the output data of task T is checkpointed, with a time c to write it onto stable storage, the total time becomes:

$$W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w+c)} - 1\right) . \tag{1}$$

Equation (1) assumes that failures can also occur during checkpoints, which is the most general model for failures. In the case of a sequence of non-checkpointed tasks to be executed on a processor P, the output data of each task resides in the memory of P for re-use by subsequent tasks. When a failure strikes P, the entire memory content is lost and the whole task sequence must be re-executed from scratch.

3.3 Problem Description and Proposed Approach

As outlined in Section 1, our objective is to not checkpoint all output data, so as to save on checkpointing overhead and thus reduce the expected overall execution time, or makespan.

The expected makespan includes the checkpointing and recovery overheads. This leads naturally to the following optimization problem: given an M-SPG to execute on processors that experience failures with a given Exponential rate, compute a schedule that does not involve direct inter-processor communications and that minimizes the expected makespan. The schedule must specify which processor executes which tasks, when each task begins execution, and which output data is checkpointed and when.

Our CkptSome approach computes a schedule that allocates tasks to processors, and that avoids direct inter-processor communications by checkpointing particular tasks so as to remove all crossover dependencies (See Section 2). It then optimally determines which additional tasks should be checkpointed so as to minimize the expected makespan.

Consider an M-SPG, G. Without loss of generality, $G = C \overrightarrow{;} (G_1 || \ldots || G_n) \overrightarrow{;} G_{n+1}$, where C is a chain and $G_1, \ldots, G_n, G_{n+1}$ are M-SPG graphs, with some of these graphs possibly empty graphs. The schedule for G is the temporal concatenation of the schedule for C, the schedule for $G_1 || \ldots || G_n$, and the schedule for G_{n+1}. A chain is always scheduled on a single processor, with all its tasks executed in sequence on that processor. When scheduling a parallel composition of M-SPGs, we use the following polynomial-time list-scheduling approach, inspired by the "proportional mapping" heuristic [15]. Given an available number of processors, we allocate to each parallel component G_i an integral fraction of the processors in proportion to the sum of the task weights in G_i (the overhead of reading/writing data to stable storage is ignored in this phase of our approach). In other terms, we allocate more processors to more costly graphs. We apply this process recursively, each time scheduling a sub-M-SPG on some number of processors. Eventually, each sub-M-SPG is scheduled on a single processor, either because it is a chain or because it is allocated to a single processor. In this case, all atomic tasks in the M-SPG are linearized based on a topological order induced by task dependencies and scheduled sequentially on the processor. This algorithm is described in Section 4.

Each time a sub-M-SPG is scheduled on a single processor, we call the set of its atomic tasks a *superchain*, because the tasks are executed sequentially even though the graph may not be a chain. We call the *entry tasks*, resp. *exit tasks*, of a superchain the tasks in the superchain that have predecessors, resp. successors, outside the superchain. Due to the recursive structure of an M-SPG, all predecessors of the entry tasks in a superchain are themselves exit tasks in other superchains. Similarly, all successors of the exit tasks in a superchain are themselves entry tasks in other superchains. This has two important consequences:

- The workflow is an "M-SPG of superchains"; and
- Checkpointing the output data of all exit tasks of a superchain means that this superchain never needs to be re-executed. In this case, we say that "the superchain is checkpointed".

A natural strategy is then to checkpoint all superchains, which avoids all crossover dependencies. More specifically, a systematic checkpoint that saves the output data of all exit tasks of a superchain is performed after the last task of that superchain completes. This checkpoint strategy is detailed in Section 5.1. Figure 5 shows an example of a schedule obtained on two processors for the M-SPG in Figure 4. A set of tasks is linearized on each processor (additional dependencies are added to enforce sequential execution of tasks on a single processor). Five checkpoints are taken: after the executions of T_1, T_{10}, T_{11}, T_{12}, and T_{13}. This guarantees that failures are contained: Once T_{13} begins executing, a failure on P_2 has no effect and a failure on P_1 is handled by immediately re-starting T_{13}.

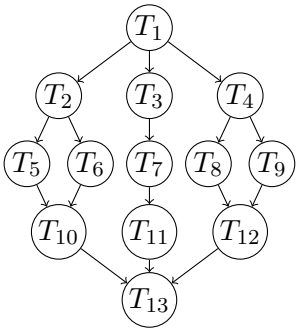


Figure 4: Example M-SPG.

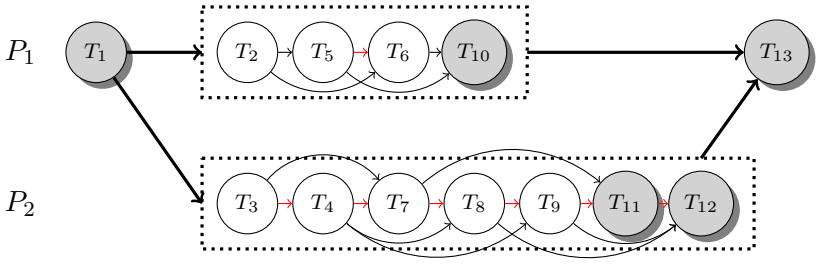


Figure 5: Mapping the M-SPG of Figure 4 onto two processors. The two superchains are shown inside boxes, with all internal and external dependencies from the original graph (red edges result from the linearization). T_{10} is the only exit task of the top superchain while T_{11} and T_{12} are the two exit tasks of the bottom superchain. A checkpoint is performed to save the output of each shadowed task.

The above approach produces a solution with the lowest number of checkpoints necessary to avoid crossover dependencies. To evaluate the expected makespan, one can then coalesce all tasks in a superchain into a single checkpointed task, leading to an M-SPG in which all tasks are checkpointed. In our example, the four tasks of the top superchain would be coalesced into one checkpointed task, and so would the seven tasks of the bottom superchain. One can then estimate the expected makespan using known algorithms for DAGs with probabilistic task durations (see Section 3.4).

While this approach avoids all crossover dependencies, and thus makes sure that failures are contained, its expected makespan may be far from optimal because too few tasks are checkpointed. Depending on the parallelism of the M-SPG and the total number of available processors, superchains may contain large numbers of tasks. If only the output data of exit tasks are checkpointed, then the expected execution time of a superchain can be large due to many re-executions from scratch. One should then checkpoint additional output data throughout the execution of the superchain. To this end, we propose a polynomial-time dynamic programming algorithm that extends the approach of Toueg and Babaoğlu [16] to determine the optimal set of output data to checkpoint. This algorithm is described in Section 5.2. Once these additional checkpoints are determined, thereby creating sequences of tasks followed by a checkpoint, we logically coalesce these sequences into a single task. Again, this is so that we can use known algorithms for estimate the expected makespan of DAGs with probabilistic task durations.

3.4 Evaluation of Expected Makespan

As discussed in Section 1, computing the expected makespan for a solution with the CkptAll strategy (tasks being already assigned to processors and all checkpointed) amounts to computing the expected longest graph of a DAG with probabilistic task durations. Recall that, once scheduled, the original workflow graph is augmented with extra dependencies to enforce serial executions of tasks at each processor.

In the DAG, task weights are random variables whose expectation is given by Equation (1). The CDF of such a random variable is complicated, because one has to account for the possibility of an arbitrary number of failures occurring at arbitrary instants. To the best of our knowledge, there is no closed-form for this CDF.

Computing the expected longest path is #P-complete, even if one considers that the execution time of a task is a discrete random variable that can take only 2 values [8]. However, basic probability theory tells us how to compute the probability distribution of the sum of two independent random variables (by a convolution) and of the maximum of two independent random variables (by taking the product of their cumulative density functions). As a result, one can compute the makespan distribution and its expected value if the DAG is a SPG (or an M-SPG), due to its recursive structure [19, 20]. However, the makespan may take an exponential number of values, which makes its direct evaluation inefficient. With only 2 values, the problem of computing the expected makespan remains NP-complete, but in the weak sense, and admits a pseudo-polynomial solution [19]. With complicated distributions for task weights as discussed above, the evaluation becomes intractable, and one has to resort to approximations.

Several approximation methods have been proposed, including approximating general graphs by series-parallel graphs [19, 20], approximating task weight distributions by Normal distributions [21, 20], or approximating the length of the longest paths [22]. Rather than using these approaches, which have various levels of accuracy depending of DAG structure, we use the classical Monte Carlo simulation approach [23, 24] with very large numbers of trials. Each trial consists in sampling the weight of each task in the DAG from its distribution. This method is compute-intensive but provides an accurate way to compare different scheduling/checkpointing strategies fairly (more accurately in practice than using the aforementioned approximation methods).

4 Scheduling M-SPGs

In this section, we describe the list-scheduling algorithm of our CkptSome approach, by which we assign sub-graphs of the workflow DAG to processors. Our algorithm decides how many processors should be allocated to parallel sub-graphs. It is recursive, so as to follow the recursive M-SPG structure, and produces a schedule of superchains, as explained in Section 3.3. It adapts the principle of "proportional mapping" heuristic [15] to M-SPGs. Pseudo-code is given in Algorithm 1.

Procedure Allocate schedules an M-SPG G, which comprises sequential atomic tasks, onto a finite set $\mathcal{P}$ of processors. It returns immediately if $G = \emptyset$ (Line 2), otherwise it decomposes G into the sequential composition of a chain, C, a parallel composition, $G_1||\dots||G_n$, and an M-SPG, G_{n+1} (Line 3). Note that several such decompositions exist and some of them lead to infinite recursions. This is the case when the chain is empty and a single graph is non-empty among $\{G_1, \dots, G_{n+1}\}$). For instance, the graph G could be decomposed such

that $C = G_1 = \ldots = G_n = \emptyset$ and $G_{n+1} = G$, or $C = G_2 = \ldots = G_{n+1} = \emptyset$ and $G_1 = G$. Our algorithm avoids these superfluous decompositions and make sure that C is the longest possible chain. It then schedules the three components in sequence. To do so, it relies on two helper procedures: the OnOneProcessor procedure, which schedules tasks on a single processor, and the PropMap procedure, when more processors are available. Allocate calls OnOneProcessor to schedule C (Line 4) and to schedule $G_1||\ldots||G_n$ if a single processor is available (Line 6). If $|\mathcal{P}| > 1$, then Allocate calls the second helper procedure, PropMap (Line 8). This procedure takes in a set of n M-SPGs and a number of processors, p, and returns a list of M-SPGs and a list of processor counts. Allocate then simply recursively schedules the i-th returned M-SPG onto a partition of the platform that contains the i-th processor count (Lines 9-12). Finally, Allocate is called recursively to schedule G_{n+1} (Line 13).

The PropMap procedure is the core of our scheduling algorithm. Let $k = \min(n, p)$ be the number of returned M-SPGs and processor counts (Line 16). Initially, the k M-SPGs are set to empty graphs (Line 17), and the k processor counts are set to 1 (Line 18). Array W contains the weight of each returned M-SPGs, initially all zeros (Line 19). Then, input M-SPGs are sorted by non-increasing weight, the weight of an M-SPG being the sum of the weights of all its atomic tasks (Line 20). Two cases are then handled. If $n \geq p$, PropMap iteratively merges each G_i with the output M-SPG that has the lowest weight so as to obtain a total of p non-empty output M-SPGs (Lines 22-25). The processor counts remain set to 1 for each output M-SPG. If instead $n < p$, then there is a surplus of processors. PropMap first assigns each input G_i to one output M-SPG (Lines 27-29). The $p - n$ extra processors are then allocated iteratively to the output M-SPG with the largest weight (Lines 30-35). Finally, PropMap returns the lists of output M-SPGs and of processor counts.

The OnOneProcessor procedure (Lines 38-41) takes as input an M-SPG and a processor, performs a topological sort of the M-SPG's atomic tasks, and then schedules these tasks, which constitute a superchain, in sequence onto the processor.

After assigning all sub-graphs of G onto processors, we complete our CkptSome approach by calling the Checkpoint procedure to decide which tasks from each superchain L to checkpoint (Lines 43-46), as described in Section 5.

5 Placing checkpoints in superchains

In this section, we describe our approach for deciding after which tasks in a superchain output data must be checkpointed. We first describe existing results for simple chains and explain how the problem is more difficult in the case of superchains. We then describe an optimal dynamic programming algorithm for superchains.

5.1 From chains to superchains

Toueg and Babaoğlu [16] have proposed an optimal dynamic programming algorithm to decide which tasks to checkpoint in a linear chain of tasks. For a linear chain, when a failure occurs during the execution of a task T, one has to recover from the latest checkpoint and re-execute all non-checkpointed ancestors of T.

In this work, we target M-SPG (sub-)graphs that are linearized on a single processor. As a result, recovery from failure is more complex than in the case of a linear chain. Consider a failure during the execution of a task T. For T to be re-executed, all its input data must

Algorithm 1 Algorithm CkptSome

procedure Allocate(G, $\mathcal{P}$)
  if $G = \emptyset$ **then return**
  $C \overrightarrow{;} (G_1 || \dots || G_n) \overrightarrow{;} G_{n+1} \leftarrow G$
  $\mathcal{L} \leftarrow$ OnOneProcessor (C, $\mathcal{P}[0]$)
  if ($|\mathcal{P}| = 1$) **then**
    $\mathcal{L} \leftarrow \mathcal{L} \cup$ OnOneProcessor ($G_1 || \dots || G_n$, $\mathcal{P}[0]$)
  else
    $(Graphs, Counts) \leftarrow$ PropMap ($G_1, \dots, G_n$, $|\mathcal{P}|$)
    $i \leftarrow 0$
    for each $graph, count$ **in** $Graphs, Counts$ **do**
      Allocate ($graph$, $\{\mathcal{P}[i], \dots, \mathcal{P}[i + count - 1]\}$)
      $i \leftarrow i + count$
  return $\mathcal{L} \cup$ Allocate (G_{n+1}, $\mathcal{P}$)

procedure PropMap($G_1, \dots, G_n$, p)
  $k \leftarrow \min(n, p)$
  $Graphs \leftarrow [\emptyset, \dots, \emptyset]$ (k elements)
  $procNums \leftarrow [1, \dots, 1]$ (k elements)
  $W \leftarrow [0, \dots, 0]$ (k elements)
  Sort $[G_1, \dots, G_n]$ by non-increasing total weight
  if $n \geq p$ **then**
    for $i = 1 \dots n$ **do**
      $j \leftarrow \arg\min_{1 \leq q \leq p} (W[q])$
      $W[j] \leftarrow W[j] + weight(G_i)$
      $Graphs[j] \leftarrow Graphs[j] \,||\, G_i$
  else
    for $i = 1 \dots n$ in G_i **do**
      $Graphs[i] \leftarrow G_i$
      $W[i] \leftarrow weight(G_i)$
    $\rho \leftarrow p - n$
    while $\rho \neq 0$ **do**
      $j \leftarrow \arg\max_{1 \leq q \leq n} (W[q])$
      $procNums[j] \leftarrow procNums[j] + 1$
      $W[j] \leftarrow W[j] \times (1 - 1/procNums[j])$
      $\rho \leftarrow \rho - 1$
  return $Graphs$, $procNums$

procedure OnOneProcessor(G, P)
  $L \leftarrow$ topological_sort(G)
  MAP (L, P) ▷ Schedule tasks serially on one processor
  return $\{L\}$

procedure CkptSome(G, $\mathcal{P}$)
  $\mathcal{L} \leftarrow$ Allocate (G, $\mathcal{P}$)
  for $L \in \mathcal{L}$ **do**
    Checkpoint (L) ▷ Decide which tasks to checkpoint

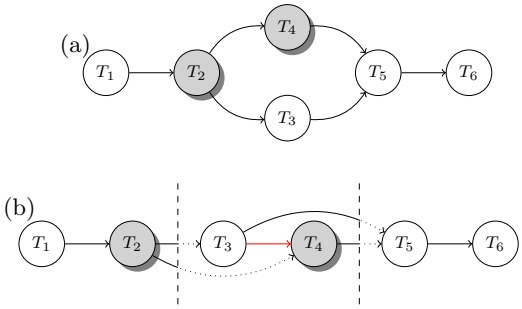


Figure 6: (a) Example M-SPG in which checkpointed tasks (T_2 and T_4) are shadowed. (b) Linearization of the M-SPG on a single processor. The dependency from T_3 to T_4, in red, results from the linearization. Vertical dashed lines correspond to checkpoints (after T_2 and T_4). Dotted lines correspond to dependencies from tasks that have been checkpointed.

be available in memory. Therefore, for each reverse path in the graph from T back to entry tasks of the superchain, one must recover from the latest checkpoint, and then recover by re-executing all non-checkpointed ancestors of T along each reverse path. Consider the M-SPG in Figure 6(a), and its linearization on a single processor in Figure 6(b). Let us assume that tasks T_2 and T_4 are checkpointed (shadowed in the figures). According to the standard definition of checkpoints, the checkpoint of T_2 includes both its output for T_3 and its output for T_4, while the checkpoint of T_4 includes only its output for T_5.

Let us now consider a single failure that occurs during the execution of T_5. To re-execute T_5, one needs to recover from the checkpointed output of T_4. But one also needs to re-execute T_3, which was not checkpointed, since the output of T_3 is needed for executing T_5. To re-execute T_3, one needs to recover from the checkpoint of T_2. This sequence of recoveries and re-executions must be re-attempted until T_5 executes successfully. As a result, the problem of deciding which tasks to checkpoint to minimize expected makespan cannot be solved by the simple linear chain algorithm in [16], which relies on a single recovery from the latest checkpoint followed by the re-execution of all tasks executed since that checkpoint.

We thus propose an alternative approach by which a checkpoint, which takes place after the execution of a task, saves not only the output data from that task, but also all non-checkpointed input data of any yet-to-be-executed task. This is shown in Figure 6, where checkpoint times are depicted as vertical dashed lines, after each execution of a checkpointed task (in this case T_2 and T_4). "Taking a checkpoint" means saving to stable storage all output data of previously executed but un-checkpointed tasks. Visually, this corresponds to solid dependence edges that cross the checkpoint time, as shown in Figure 6. With this extended definition of checkpoints, the checkpoint after T_4 now includes the output data of T_3 for T_5, in addition to the output of T_4 for T_5. This approach allows the algorithm in [16] to be extended to the case of superchain as described in the next section.

5.2 Checkpointing algorithm

To answer the question of when to take checkpoints throughout the execution of a superchain on a processor, we propose an $O(n^2)$ dynamic programming algorithm. For each sequence of tasks allocated to a processor, the algorithm finds the optimal set of tasks after which output

data must be checkpointed in order to minimize its expected completion time. For each task of a sequence, it determines the position of the last checkpoint that optimizes the expected completion time. The set of tasks after which a checkpoint is taken can then be obtained by backtracking from the last task of the superchain.

Let us consider a superchain that contains tasks $T_a, \ldots, T_b$ (we assume that tasks $T_1, \ldots, T_n$ are numbered according to a topological sort in such a way that tasks from any superchain have contiguous indices). Without loss of generality let us assume that T_j executes immediately before T_{j+1}, $j = a, \ldots, b-1$ and that T_a starts as soon as the necessary input data is read from stable storage.

Our approach always takes a checkpoint after T_b completes to avoid crossover dependencies (see Section 3.3), thus ensuring that all output data from all exit tasks of the superchain are checkpointed.

Let $\mathcal{E}Time(j)$ be the optimal expected time to successfully execute tasks $T_a, \ldots, T_j$, when a checkpoint is taken immediately after T_j completes (with possibly earlier checkpoints). Our goal is to minimize $\mathcal{E}Time(b)$. To compute $\mathcal{E}Time(j)$, we formulate the following dynamic program by trying all possible locations for the last checkpoint before T_j:

$$\mathcal{E}Time(j) = \min\left(T(a,j), \min_{a \le i < j}\{\mathcal{E}Time(i) + T(i+1,j)\}\right),$$

where $T(i+1,j)$ is the expected time to successfully execute tasks T_{i+1} to T_j, provided that a checkpoint occurs after task T_j completes and the previous checkpoint occurred before task T_{i+1} starts. This account for the time to read the input data, execute the tasks and perform the checkpoint. As there is no checkpoint between tasks T_{i+1} and T_j, all intermediate data are kept in memory and retrieved instantly. This reduces the checkpoint overhead compared to CkptAll.

According to Equation (1), the expected time needed to execute tasks T_i to T_j for each (i,j) pair with $i \le j$ is given by

$$T(i,j) = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(R_i^j + W_i^j + C_i^j)} - 1\right), \tag{2}$$

where λ is the processor's exponential failure rate, R_i^j is the time necessary to read from stable storage all data produced by tasks $T_1, \ldots, T_{i-1}$ and needed by tasks $T_i, \ldots, T_j$, $W_i^j = w_i + \ldots + w_j$ is the time to execute tasks T_i to T_j when no failures occur, C_i^j is the time taken to checkpoint the input data of $T_{j+1}, \ldots, T_n$ that is produced by $T_i, \ldots, T_j$ (i.e., the non-checkpointed predecessors of $T_{j+1}, \ldots, T_n$ in $T_i, \ldots, T_j$), and d is the downtime. Formally, $R_i^j = \sum_{k=i}^{j} \sum_{T_l \in \mathrm{Pred}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{lk}$ and $C_i^j = \sum_{k=i}^{j} \sum_{T_l \in \mathrm{Succ}(T_k) \setminus \{T_i, \ldots, T_j\}} c_{kl}$ where c_{kl} is the cost to read or write the data produced by T_k and needed by T_l, $\mathrm{Pred}(T_k)$ is the set of predecessors of T_k and $\mathrm{Succ}(T_k)$ is the set of successors of T_k. Note that the data that is read (during R_i^j) may be produced by exit tasks of previous superchains and that the data that is saved (during C_i^j) may be needed by entry tasks in next superchains. In particular, C_i^j is greater than or equal to the time to checkpoint all output data of T_j.

The pseudo-code for this dynamic programming solution is given in Algorithm 2. The computation of $\mathcal{E}Time(j)$ takes $O(n)$ time, as it depends on at most j other entries. The computation of $T(i,j)$ for all (i,j) pairs with $i \le j$ takes $O(n^2)$ time. Therefore, the overall complexity is $O(n^2)$.

Algorithm 2 CHECKPOINT

```
1: procedure CHECKPOINT(T_a, ..., T_b)
2:     last_ckpt ← [0, ..., 0] (b − a + 1 elements)
3:     for j = a ... b do
4:         ETime(j) ← T(a, j)
5:         last_ckpt[j] ← 0
6:         for i = a ... j − 1 do
7:             temp ← ETime(i) + T(i + 1, j)
8:             if temp < ETime(j) then
9:                 ETime(j) ← temp
10:                last_ckpt[j] ← i
11:    Ckpts ← ∅                                  ▷ List of tasks to checkpoint
12:    while b ≠ a do                             ▷ Backtracking
13:        Ckpts ← Ckpts ∪ {T_b}                  ▷ Checkpoint after task T_b
14:        b ← last_ckpt[b]
15:    return Ckpts
```

5.3 Technical remarks

Remark #1 – Our model assumes that faults may occur while reading or writing data to stable storage. We could also use the simpler assumption that faults only occur during computations, as done in many previous works, by replacing Equation (2) by $(\frac{1}{\lambda}+d)(e^{\lambda W_i^j}-1)+R_i^j+C_i^j$.

Remark #2 – It may appear wasteful to read files from stable storage that were just written by the same processor and that may still be in memory or accessible locally on disk. An alternative strategy would be to assume that the processor still has access to each data item that it has computed before (until a failure strikes). The initial read time would then be reduced to $R_i^j - \sum_{k=i}^{j} \sum_{T_l \in \text{Pred}(T_k) \cap \text{Alloc}(T_i) \setminus \{T_i,\dots,T_j\}} c_{lk}$ where $\text{Alloc}(T_i)$ is the set of tasks allocated to the same processor as T_i (i.e., we no longer include the task predecessors that were on the same processor). However, the recovery cost should also include the time to read all these data back from stable storage in case of failure. This cost would be $R_i^j + \sum_{k=1,T_k \in \text{Alloc}(T_i)}^{i-1} \sum_{T_l \in \text{Succ}(T_k) \cap \text{Alloc}(T_i) \setminus \{T_1,\dots,T_j\}} c_{kl}$. Note that in case of multiple failures on the same processor, data may be read back from stable storage more than once, which is also wasteful (it would be more efficient to recover data whenever they are needed instead), but this overestimation of the recovery cost is necessary to apply our dynamic programming approach. For low failure probability, it may thus be advantageous to use this alternative strategy, because the higher recovery cost is offset by the lower initial cost. We did not explore this option further, but our method can be easily extended to encompass it.

Remark #3 – Algorithm 2 can be further improved by adjusting the checkpointing costs of files that are systematically saved (the outputs of exit tasks that are required by other superchains). When these files are large compared to the others, the dynamic programming approach may lead to fewer checkpoints than with small such files because each aggregated checkpoint cost (C_i^j) is large. However, these files are always checkpointed (to avoid crossover dependencies) and should have no impact on where to place additional checkpoints. A solution is to integrate the cost of each such checkpoint (c_{kl}) into the cost of its producing task (w_k), assuming that this checkpoint is done directly after the execution of the task, and to discard these costs from the aggregated checkpoint costs (C_i^j). This optimization is particularly useful

when these necessary checkpoints are costly.
Remark #4 – We said that a superchain is checkpointed when the output data of all its exit tasks are checkpointed. However, this does not mean that these output data need to be checkpointed after the execution of the last task of the superchain. Consider the superchain in the example of Figure 5 with two exit tasks T_{11} and T_{12}. Algorithm 2 systematically takes a checkpoint after the last task T_{12} but not necessarily after T_{11}. If a checkpoint is taken after T_{11}, then its output data is saved before T_{12} executes. Otherwise, this output data is saved when T_{12} completes. Both options are possible. Regardless, the structure of M-SPGs ensures that T_{11} and T_{12} have the same successors outside the superchain, and thus recovery is straightforward.

6 The CkptNone strategy

In this section we establish the complexity of computing the expected makespan of a scheduled task graph when the CkptNone strategy is used. In Section 6.1 we construct a simple instance and show that it is already #P-complete, thereby establishing the #P-completeness of the problem. Then in Section 6.2 we derive a simple formula to approximate the expected makespan.

6.1 #P-completeness

Let us defined the following problem:

Definition 1 (DAG-mks)**.** Consider a task graph with n tasks. Each task T_i is scheduled on its own processor P_i and has a unitary cost. Each task can thus start executing as soon as all its predecessors have completed (there are no resource constraints). There is a fixed probability p_i that each processor P_i fails when it executes its allocated task T_i for $1 \leq i \leq n$. Once P_i has failed, it restarts at the next time-step and it cannot fail again. Hence, if P_i fails while executing T_i, it will successfully re-execute T_i during the next time-step. The problem is to compute the expected makespan of the schedule.

In this simplified problem, we have discrete times-steps, and failures hit processors only once, similarly to the approximated execution model given in Equation (1). Note that with this simple model, the schedule is always executed in bounded time.

Theorem 1. DAG-mks *is #P-complete.*

Proof. We show this result with a reduction from REL [8, 11], a #P-complete problem. Consider a DAG with a source vertex, and let V_i be the set of vertices with a path of length $i-1$ from the source. In the following, we consider layered graphs, and V_i is thus the set of vertices on layer i. A transportation DAG is a graph in which edges go only from the source $v_1 \in V_1$ to vertices in V_2, from vertices in V_2 to vertices in V_3 and from vertices in V_3 to the sink $v_n \in V_4$. In other words, this is a four-layer graph shaped as a directed bipartite graph with a source and a sink (see Figure 7).

Definition 2 (REL)**.** We consider a transportation DAG with possibly multiple edges and where each edge may fail with probability p. The objective is to determine the probability that there is a path between the source and the sink.

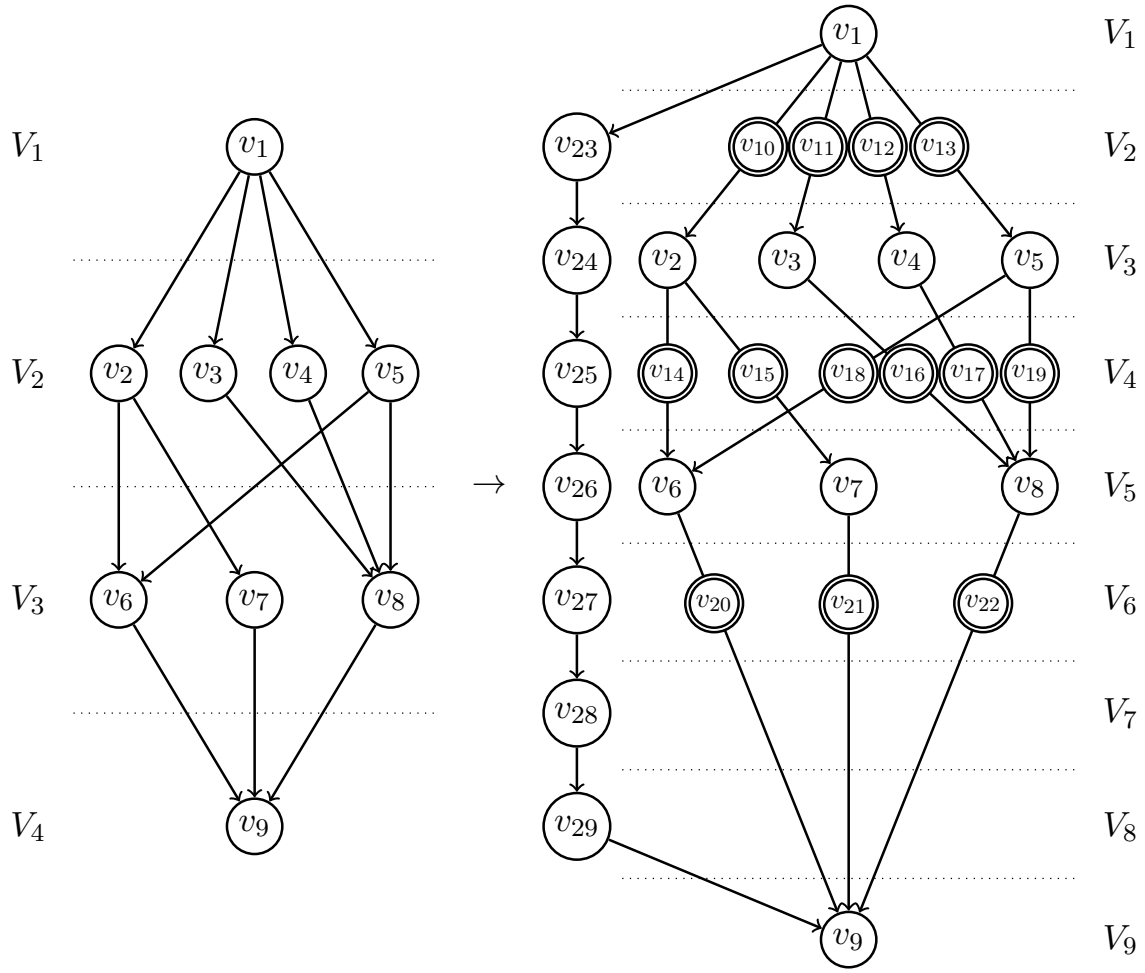


Figure 7: The transportation graph of the REL instance (left) and the corresponding DAG-MKS instance (right). In the REL instance, each edge may fail with probability p. In the DAG-MKS instance, tasks with a double circle (v_{10} to v_{22}) may fail with probability $1-p$, while other tasks never fail.

We first transform an instance of REL into an instance of a related problem in which the vertices fail instead of the edges. Each initial vertex remains unchanged and cannot fail. We replace each edge by a vertex that can fail with probability p and connect this vertex to the predecessor and the successor of the edge. This leads to a transformed graph with 7 layers of vertices. Vertices in even layers fail with probability p, whereas vertices in odd layers do not fail. The probability that there is a path between the source and the sink is the same as with the initial REL instance.

We now build an instance of DAG-MKS with the same graph structure, and we let $p_i = 1-p$ for all vertices of even layers and $p_i = 0$ otherwise. We will prove that determining the probability that the makespan of this DAG is equal to 10 solves the REL instance.

We introduce some notations for the REL instance. Let E_{ij} be the event that occurs when the edge from vertex v_i to v_j succeeds ($\Pr[E_{ij}] = 1-p$). All E_{ij} are independent. Let F_i^j be the event that occurs when there is a path from the source to a vertex $v_i \in V_j$ in the REL

instance. Then, F_1 always occurs, $F_i = E_{1i}$ for $v_i \in V_2$, $F_i = \bigcup_{j \in \text{Pred}(v_i)} F_j \cap E_{ji}$ for $v_i \in V_3$ and $F_n = \bigcup_{j \in \text{Pred}(v_n)} F_j \cap E_{jn} = \bigcup_{j \in \text{Pred}(v_n)} \bigcup_{k \in \text{Pred}(v_j)} E_{1k} \cap E_{kj} \cap E_{jn}$ (recall that $\text{Pred}(v_i)$ is the set of predecessors o v_i). Solving REL consists in determining $\Pr[F_n]$.

We now focus on the DAG-MKS instance. Let G_i be the event that occurs when vertex v_i in layer V_j fails at step j and is re-executed, for $j \in \{2, 4, 6\}$ (recall that vertices in odd layers never fail). We have $\Pr[G_i] = 1 - p$, which is equivalent to the event $E_{\text{pred}(v_i)\text{succ}(v_i)}$ (we use pred, resp. succ, in lowercase to denote a single predecessor, resp. successor). All G_i are independent. Let C_i be the completion time of vertex v_i. Consider the first three layers. The event $\{C_1 = 1\}$ always occurs, because the source vertex never fails. For $v_i \in V_2$, either no fault occurs ($\overline{G_i}$) and $C_i = 2$, or a fault occurs and it takes one more time-step to execute task v_i, i.e., we derive that $G_i = \{C_i = 3\}$. Finally, $\{C_i = 4\} = \{C_{\text{pred}(v_i)} = 3\} = G_{\text{pred}(v_i)}$ for $v_i \in V_3$. Analogously, for the two next layers, we have: $\{C_i = 6\} = \{C_{\text{pred}(v_i)} = 4\} \cap G_i$ for $v_i \in V_4$ and $\{C_i = 7\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_j = 6\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_{\text{pred}(v_j)} = 4\} \cap G_j$ for $v_i \in V_5$. For the last two layers, we have: $\{C_i = 9\} = \{C_{\text{pred}(v_i)} = 7\} \cap G_i$ for $v_i \in V_6$ and $\{C_n = 10\} = \bigcup_{j \in \text{Pred}(v_i)} \{C_j = 9\} = \bigcup_{j \in \text{Pred}(v_n)} \{C_{\text{pred}(v_j)} = 7\} \cap G_j$. After simplification, we have $\{C_n = 10\} = \bigcup_{j \in \text{Pred}(v_n)} \bigcup_{k \in \text{Pred}(\text{pred}(v_j))} G_{\text{pred}(\text{pred}(v_k))} \cap G_k \cap G_j$. We see that $\Pr[\{C_n = 10\}] = \Pr[F_n]$ because the graph structure of the DAG-MKS instance is the same as REL.

It remains to prove that determining the probability that the makespan is 10 (i.e., $\Pr[\{C_n = 10\}]$) can be done by determining the expected makespan. We use a technique similar to the one used in [8]. We simply add a series of 7 never-failing vertices between the source and the sink, in parallel of the previous graph (see Figure 7). Then, the expected makespan of this new DAG is $\Pr[\{C_n = 10\}] + 9$. □

The general problem (i.e., when task costs are not unitary, when several tasks may be allocated to a given processor, when there is a probability of failure during re-execution, when there are recovery costs, etc.) is thus also #P-complete, and likely even more challenging than DAG-MKS.

6.2 Approximating the makespan

Section 6.1 shows the difficulty of computing the makespan of a schedule where no task is checkpointed. Still, we can derive the following approximation:

Theorem 2. *Consider a schedule for an* M-SPG *G with p processors, with all tasks assigned to processors and no checkpoint. Let W_{par} be the parallel time of the schedule with no failure, and let λ be the processor's exponential failure rate. An approximation of the expected makespan $EM(G)$ is*

$$EM(G) = \left(\frac{1}{p\lambda} + d\right)(e^{p\lambda W_{par}} - 1)$$

Proof. The idea is to consider a single task of weight W_{par} and to compute its expected execution time as in Equation (2). The only differences are that: (i) we use the platform's Exponential failure rate $p\lambda$ [18]; and (ii) we neglect the recovery cost. □

While this formula is likely to be inaccurate, we are not aware of any better approximation. In Section 7, we do not use $EM(G)$ to evaluate the expected makespan of the CkptNone strategy; instead, we use Monte-Carlo simulations. We consider a single task of weight W_{par}

and compute its expected execution time by sampling its exponential distribution with failure rate $p\lambda$. After repeating this operation for a large number of trials, each of which produced a sample makespan, we approximate the expected makespan as the average over these samples, thereby obtaining an accurate evaluation.

7 Experiments

In this section, we present experimental results that quantify the effectiveness of the proposed CkptSome algorithm.

7.1 Experimental methodology

Our experiments are for representative workflow applications generated by the Pegasus Workflow Generator (PWG) [25, 26, 17]. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows (the parameters of which, e.g., total number of tasks, can be chosen). We consider three different classes of workflows generated by PWG, namely Montage, Ligo and Genome, which are all M-SPGs[2] (information on the corresponding scientific applications is available in [17, 27]):

- Montage: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a Montage task is 10s. Structurally, Montage is a three-level graph [28]. The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists in a join followed by a fork. Then, the third level (co-addition to form the final mosaic) is simply a join.
- Ligo: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems. The average weight of a Ligo task is 220s. Structurally, Ligo can be seen as a succession of Fork-Joins meta-tasks, that each contains either fork-join graphs or bipartite graphs (see the LIGO IHOPE workflow in [17]). Depending on the number of tasks required, PWG may not output an M-SPG Ligo workflow because of some incomplete bipartite graphs. In these cases, to ensure full fairness when comparing approaches, the baseline strategies process the original workflow while CkptSome processes a workflow where bipartite graphs have been extended with dummy dependencies for zero-size files (which adds synchronizations but no data transfers).
- Genome: The epigenomics workflow created by the USC Epigenome Center and the Pegasus team automates various operations in genome sequence processing. The average weight of a Genome task depends on the total number of tasks and is greater than 1000s. Structurally, Genome starts with many parallel fork-join graphs, whose exit tasks are then both joined into a new exit task, which is the root of fork graphs (see the Epigenomics workflow in [17]).

We generate Montage, Ligo, and Genome workflows with various number of tasks. For each task T_i in the workflow, its weight w_i is generated by PWG. We compute the time

[2] Montage is not fully an M-SPG because of some transitive edges that go from the source tasks to the exit tasks of the second layer. However, this does not impact CkptSome because the source tasks are also exit tasks and are thus always checkpointed.

required to read or save the data produced by task T_i and needed by task T_j, c_{ij}, by dividing the size of the file in bytes by the stable storage bandwidth in byte/sec. The file sizes are generated by PWG. In some instances, a task may generate the same file for more than one successor task. In this case a checkpoint saves the file only once.

In the experiments we consider different exponential processor failure rates. To allow for consistent comparisons of results across different M-SPGs (with different numbers of tasks and different task weights), we simply fix the probability that a task fails, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given M-SPG $G = (V, E)$ and a given p_{fail} value, we compute the average task weight as $\bar{w} = \sum_{i \in V} w_i / |V|$, where w_i is the weight of the i-th task in V. We then pick the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda \bar{w}}$. We conduct experiments for three p_{fail} values: 0.01, 0.001, and 0.0001.

An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the data-intensiveness of the application. The workflows generated by PWG give task durations in seconds and file sizes in bytes, which makes it difficult to quantify data-intensiveness. Instead, we define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. The total store time is the total file size divided by the bandwidth to the stable storage. Instead of picking arbitrary bandwidth values, which would have different meanings for different workflows, we vary the CCR by scaling file data sizes by a factor. This makes it possible to consider and quantify the data-intensiveness of all workflows in a coherent manner across experiments and workflow classes and configurations.

The experiments compare CkptSome to the two extreme approaches, CkptAll and CkptNone. For all strategies, we use Monte-Carlo simulations [23, 24] to compute the expected makespan of the solutions. A task in the DAG succeeds or fails as determined by sampling the exponential time-to-failure distribution, and a task can fail more than once. After sampling, the DAG is deterministic and its makespan can be computed as the length of its longest path. This operation is repeated for a large number of trials, each of which produces a sample makespan. These samples approach the actual makespan distribution as the number of trials increases. Following [22], we use 300,000 trials and approximate the expected makespan as the average over the resulting 300,000 makespan samples. This enormous number of trials is prohibitively expensive in practice, but provides us with an accurate ground truth to compare the different strategies. Code is publicly available at [29].

7.2 Expected makespan

In this section, we compare the expected makespan of two baseline strategies (CkptAll and CkptNone) over that of our proposed strategy (CkptSome). Figures 8, 9, and 10 show expected makespans for CkptAll and CkptNone relative to that of CkptSome vs. Communication-to-Computation Ratio (CCR). Data points above the $y = 1$ line denote cases in which our strategy outperforms a competitor (i.e., achieves a lower expected makespan). Figures 8 and 9 show results for Genome and Montage with 50, 100, 300, 500, 700 and 1000 tasks, figure 10 shows results for Ligo with 50, 100, 300, 400 and 1000 tasks, for various numbers of processors P, and for the three p_{fail} values (0.01, 0.001, and 0.0001). We report the average and maximum number of failures that occur for the 300,000 trials of each execution. These numbers are shown above the horizontal axis in each figure, labeled as *avg* and *max*. For different workflows, we pick the number of processors, P, as follows: we compute the

maximum parallelism of the workflow, p, and pick $P \in \{p/4, p/2, 3p/4, p\}$.

A clear observation is that CkptSome always outperforms CkptAll.[3] In each scenario, above some CCR value, which depends on the failure rate and the workflow size, CkptSome leads to significant improvement over CkptAll. As the CCR decreases, the relative expected makespan of CkptAll decreases and converges to 1. This is because when checkpointing becomes cheap enough CkptSome decides to checkpoint every task, and thus is equivalent to CkptAll.

Another common trend is that the relative expected makespan of CkptNone increases as the CCR decreases since as checkpoints become cheaper not checkpointing becomes a losing strategy (poorer resilience to failures, but little saving on checkpointing overhead). Overall, CkptNone becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the rightmost column to the leftmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom left corner of the figures), the relative expected makespan of CkptNone is so high that it does not appear in the plots.

CkptSome achieves better results than CkptNone except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet CkptSome by design always checkpoints some tasks (it checkpoints all exit tasks of superchains). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. The results above for our particular benchmark workflows, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in particular practical situations.

8 Related work

Checkpointing workflows has received considerable attention in the recent years, but no satisfactory solution has been proposed for fail-stop failures and general DAGs. For completeness we first review related work devoted to soft errors (Section 8.1). We then review work devoted, like this work, to fail-stop errors (Section 8.2).

8.1 Soft and silent errors

Many authors have considered soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. Checkpointing, or more precisely making a copy of all task input/output data, is the most widely used technique to address soft errors. If a soft error occurs during its execution, a task can then be re-executed from scratch. This solution can be too costly, and it is possible to save a copy of task input/output data only periodically, at the price of more re-execution when an error is detected. This is the trade-off analyzed by Cao et al. [31] for Cholesky factorization. Several authors have suggested techniques that identify tasks on the critical path, and then making scheduling decisions that attempt to ensure the timely execution of these tasks [32, 33]. A widely used technique to cope with soft errors is task replication, the challenge being to avoid over-duplicating tasks so

[3] There are in fact a couple of CCR values for Ligo with 300 and 400 tasks for which this is not true. This is an artifact of our slight transformation of the Ligo workflow (see Section 7.1 for details).

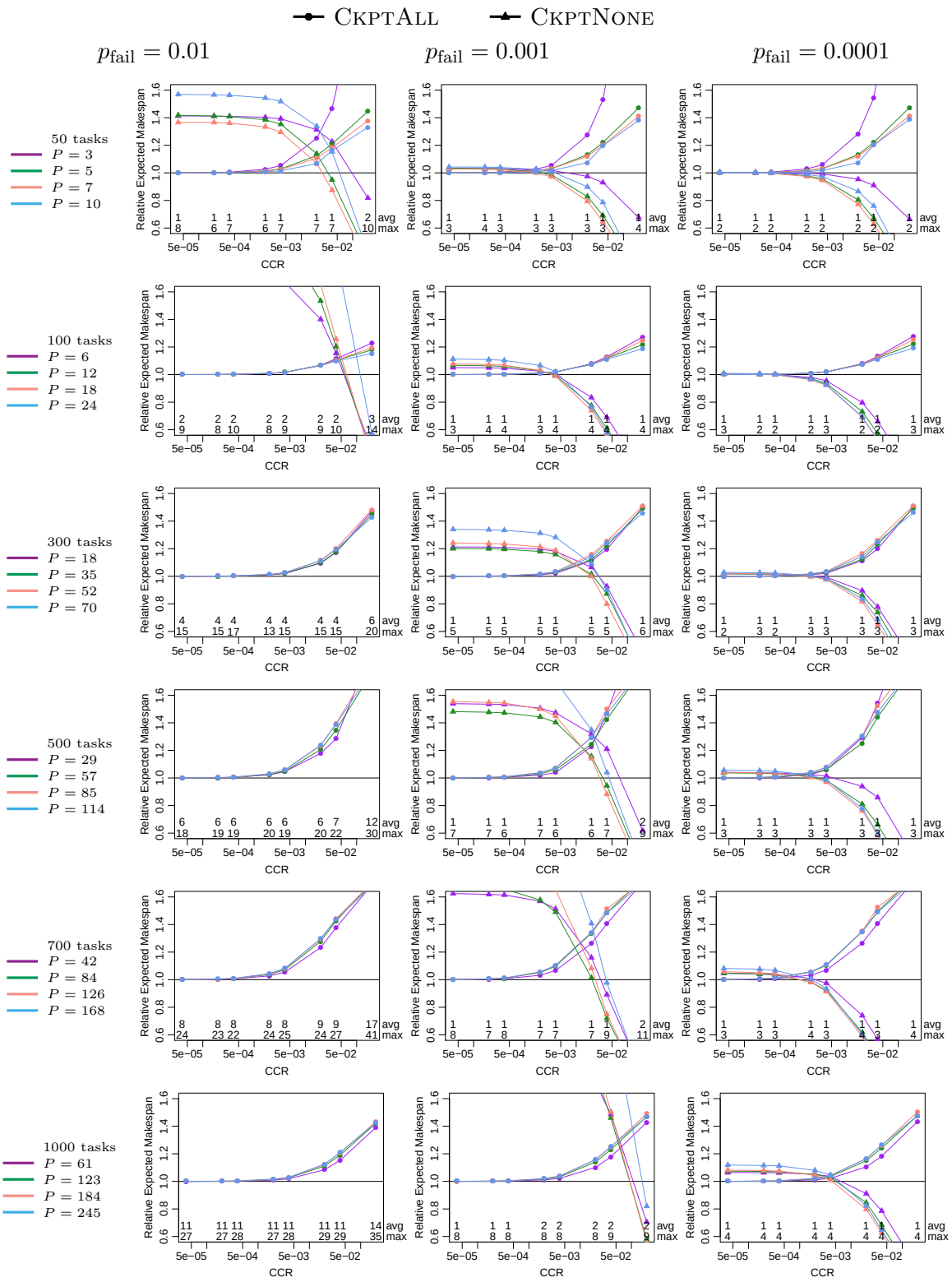


Figure 8: Expected makespan of CkptAll and CkptNone relative to that of CkptSome for the Genome workflow, three different failure rates, six workflow sizes, and varying Communication-to-Computation Ratio (CCR).

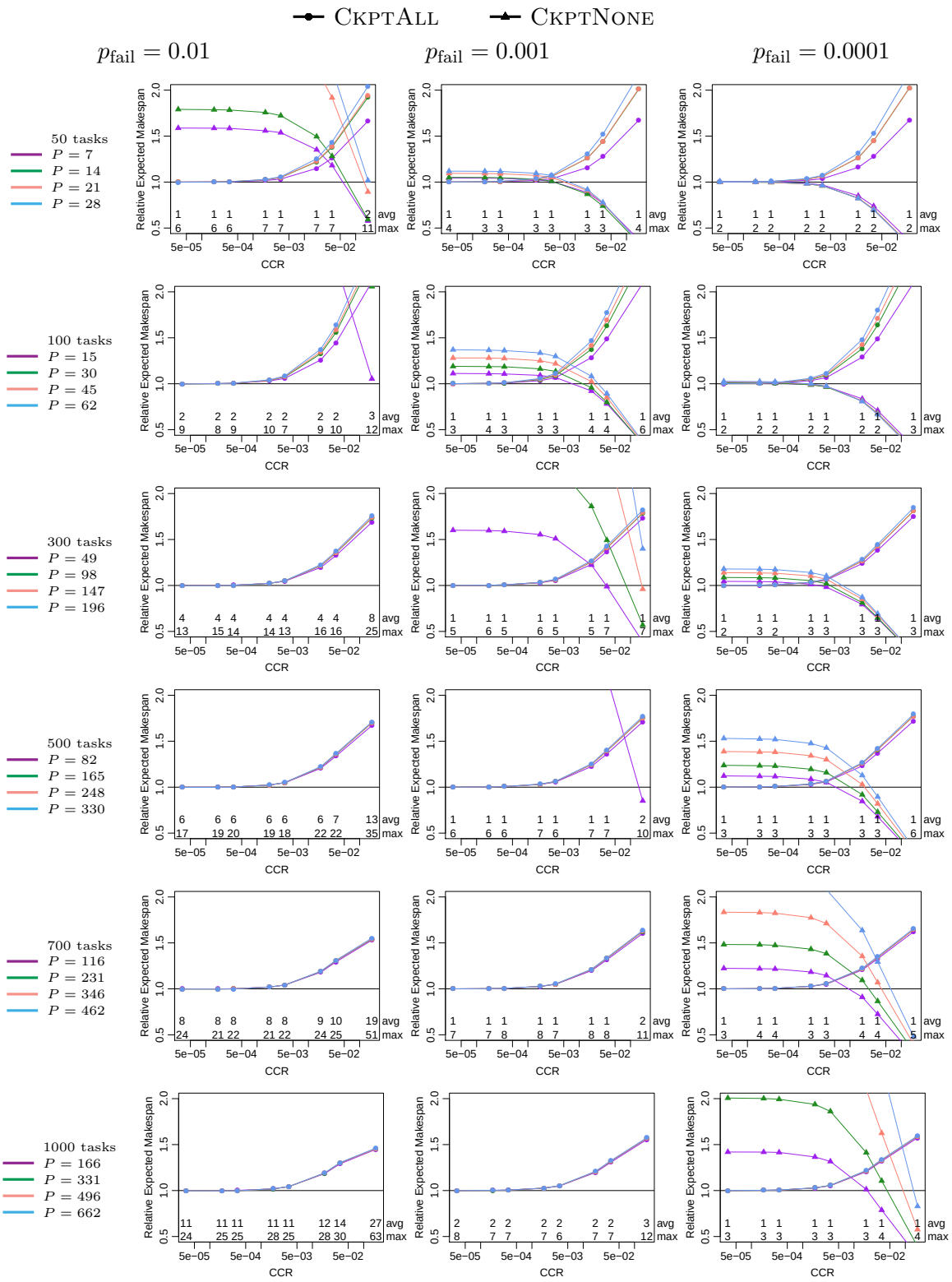


Figure 9: Expected makespan of CkptAll and CkptNone relative to that of CkptSome for the Montage workflow, three different failure rates, six workflow sizes, and varying Communication-to-Computation Ratio (CCR).

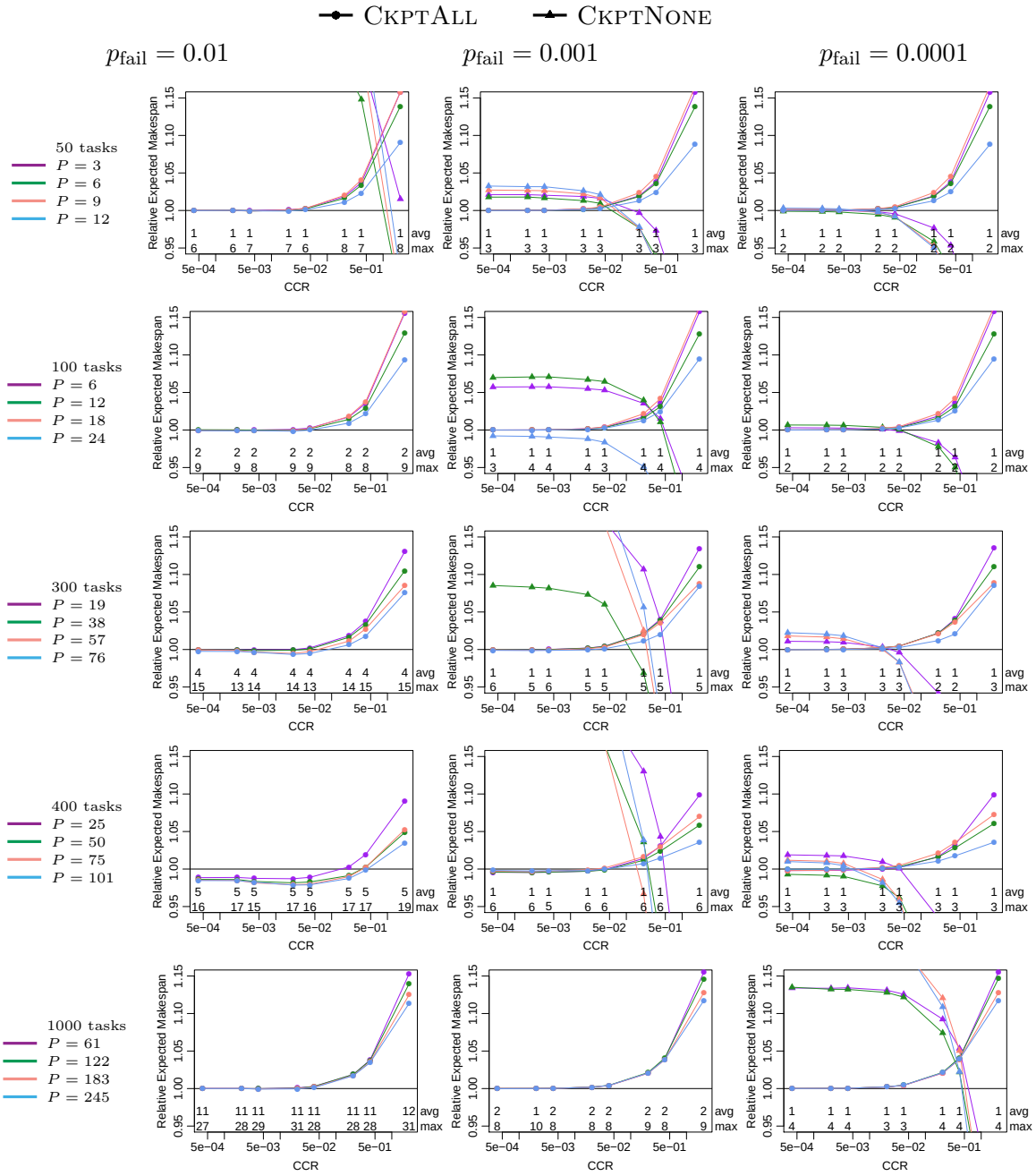


Figure 10: Expected makespan of CkptAll and CkptNone relative to that of CkptSome for the Ligo workflow, three different failure rates, five workflow sizes, and varying Communication-to-Computation Ratio (CCR).

as to strike a good balance between fast failure-free executions and resilient executions [34]. Two representative practical frameworks are the NARBIT system [35], which recovers from soft errors via task replication and work stealing, and Nanos [36, 37], a runtime system that supports the OpenMP programming model.

Silent errors represent a different challenge than soft errors, in that they do not interrupt the execution of the task but corrupt its output data. However, their net effect is the same, since a task must be re-executed whenever a silent error is detected. A silent error detector is applied at the end of a task's execution, and the task must be re-executed from scratch in case of an error. Checkpointing (making copies of input/output data) or replicating tasks and comparing outputs, are two common techniques to mitigate the impact of silent errors. With checkpointing, several application-specific detectors can be used to avoid replication and increase performance in failure-free executions. Two well-known examples are Algorithm-Based Fault Tolerance (ABFT) [38, 39, 40] and silent error detectors based on domain-specific data analytics [41, 42, 43].

Several work exist that attempt to provide resilience to arbitrary DAGs in the presence of silent errors. All of them are based on some task replication mechanism. Hashimoto et al. [44] propose two multiprocessor scheduling algorithms for arbitrary DAGs, but they can only work on systems victim of at most one single silent error. The other work we are aware of try to maximize reliability, that is, the probability that the application execution is not victim of a single failure. Girault and Kalla [45] propose an exponential-time algorithm for bi-criteria multiprocessor scheduling which returns a static schedule for the input DAG under upper bound constraints on the application execution time and on the *global system failure rate*. Subasi et al. [46] use partial replication to improve the reliability of an application in presence of silent and fail-stop errors. Works that optimize reliability do not guarantee that all executions will eventually succeed (because, for instance, not all failure patterns are covered by the chose replication scheme). By contrast, works, like this one, that optimize the expectation of the makespan guarantee that all executions successfully complete (otherwise, the expectation of the makespan would be infinite!).

8.2 Fail-stop failures

By contrast with soft and silent errors, relatively few published works have studied fail-stop failures in the context of workflow applications.

Consider first a workflow that consists of a linear chain of tasks. The problem of finding the optimal checkpoint strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time, has been solved by Toueg and Babaoglu [16] using a dynamic programming algorithm. Note that the tasks can themselves be parallel, but the execution flow is sequential, which dramatically limits the amount of re-execution in case of a failure. The algorithm of [16] was later extended in [47] to cope with both fail-stop and silent errors simultaneously.

Consider now a general workflow comprised of parallel tasks that each executes on the whole platform. Therefore, the workflow execution is linearized, and in essence reduces to a chain of macro-tasks executing on a single macro-processor, whose speed is the aggregate speed of the available processors, and whose failure rate is proportional to the number of available processors. Checkpoints can then be placed after some tasks. However, because the original workflow is not a chain, it is more complicated to keep track of live output data, and the problem of placing checkpoints is NP-complete for simple join graphs [48]. To circumvent

this problem, when checkpointing a task, one can decide to checkpoint not only the task's own output data, but also all the live data that will be needed later on in the workflow. This is the main idea of the algorithm proposed in Section 5.

Finally, consider a general workflow whose tasks do not span the whole platform when executing. Existing work in this most general context diverges from ours as follows: either there is a limit to the number of failures that an execution can cope with, or the optimization objective is reliability, meaning that application execution can fail.

Limiting the number of possible failures renders the problem more tractable (and is done also in the context of silent errors [44]). For instance, Wang et al. [49] present a replication-based approach, called Imitator, which is only guaranteed to succeed when no more than k fail-stop failures occur during a given execution of a DAG (which is executed repeatedly), assuming that there are $k + 1$ replicas.

In terms of works that target application reliability, is the work by Assayad et al. [50] on multi-criteria scheduling for real-time systems. They try to simultaneously minimize the application makespan and the probability that an execution succeeds. Jacques-Silva et al. describe in [51] a modeling framework for evaluating the dependability of streaming applications under faults that lead to data loss or silent data corruption. This framework is used to compare three fault tolerance techniques, including checkpointing. However, in their model, even checkpoints do not guarantee a successful application execution.

To the best of our knowledge, this work is the first approach (beyond application-specific solutions) that does not resort to linearizing the entire workflow as a chain of (macro-)tasks, that can cope with an arbitrary number of failures, that always guarantee a successful application execution, and that minimizes the (expectation of) the application execution time. As a result, we propose the first DAG scheduling/checkpointing algorithm that allows independent (sequential) tasks to execute concurrently on multiple failure-prone processors in standard task-parallel fashion.

9 Conclusion

We have proposed a scheduling/checkpointing algorithm, called CkptSome, for executing workflow applications on parallel computing platforms in which processors are subject to fail-stop failures. The objective function to be minimized is the expectation of the makespan, which is a random variable due to probabilistic task re-executions due to failures. For general Directed Acyclic Graphs (DAGs), this problem is intractable and even computing the objective function is itself a #P-complete problem. However, by restricting our work to a class of structured recursive DAGs, Minimal Series-Parallel Graphs (M-SPGs), which are broadly relevant to production workflow applications, we are able to design a sensible algorithm and to accurately compute the expected makespan of the solutions it produces. A competing approach, CkptAll, side-steps part of the difficulty of solving the problem by saving all application data to stable storage so as to minimize the impact of failures, with the drawback of maximizing checkpointing overhead. This is the approach employed by default in most production workflow executions, in which each task is an executable that reads all its input from files and writes all its output to files. Another competing approach, CkptNone, is a risky zero-overhead approach in which the entire workflow is re-executed from scratch in case of a failure. The broad objective of our algorithm is to produce solutions that strike a good compromise between these two extremes. Note that for the CkptNone approach, when

applied to general DAGs, we have established that the problem of computing the expected makespan is #P-complete, which to the best of our knowledge is a new result.

We have evaluated the effectiveness of our algorithm by considering realistic workflow configurations produced by a workflow generator from the Pegasus community [25, 26, 17]. We have shown that our CkptSome algorithm does indeed provide an attractive compromise between the CkptAll and CkptNone approaches. More specifically, CkptSome always outperforms CkptAll and is only outperformed by CkptNone when checkpoints are expensive and/or failures are rare. Our experimental methodology provides the quantitative means to identify these cases (based on application CCR, platform scale, and failure rates), so as to select which approach to use in practice.

Future work will be devoted to extending the scheduling algorithms to parallel (moldable) tasks, and to deriving graph transformation techniques to enable the approach to arbitrary workflows. We point out that CkptSome can be straightforwardly extended to deal with General Series Parallel Graphs, which are defined in [13] as graphs whose transitive reductions are M-SPGs.

Another promising direction is to refine the linearization algorithm for superchains (Algorithm 1). Instead of choosing the topological sort arbitrarily, one may try and reduce the total volume of output files, in the hope of reducing the total checkpointing cost when applying Algorithm 2 after the linearization. This problem is related to the sum cut problem [52], which is NP-complete for general DAGs, but may be amenable to efficient solutions for M-SPGs.

Finally, this work decouples the allocation of tasks to processors and the checkpointing decisions. While the final part of the checkpointing stage is optimal, it may be possible to improve our overall approach by making allocation and checkpointing decisions simultaneously. This is a challenging proposition, and for now a solution seems out of reach.

Acknowledgments

We would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper. This work is supported in part by NSF award SI2-SSE:1642369, and by the PIA ELCI project.

References

[1] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, no. 0, pp. 17–35, 2015.

[2] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong *et al.*, "Askalon: A development and grid computing environment for scientific workflows," in *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[3] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[4] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud," *Nucleic acids research*, p. gkt328, 2013.

[5] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Proc. 16th Int. Conf. Scientific and Statistical Database Management*. IEEE, 2004, pp. 423–424.

[6] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 1.

[7] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform," in *Proc. 26th IEEE Int. Parallel and Distributed Processing Symposium*, 2012, pp. 1352–1363.

[8] J. N. Hagstrom, "Computational complexity of PERT problems," *Networks*, vol. 18, no. 2, pp. 139–147, 1988.

[9] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 5th ed. Springer, 2016.

[10] L. G. Valiant, "The complexity of enumeration and reliability problems," *SIAM J. Comput.*, vol. 8, no. 3, pp. 410–421, 1979.

[11] J. S. Provan and M. O. Ball, "The complexity of counting cuts and of computing the probability that a graph is connected," *SIAM J. Comp.*, vol. 12, no. 4, pp. 777–788, 1983.

[12] H. L. Bodlaender and T. Wolle, "A note on the complexity of network reliability problems," *IEEE Trans. Inf. Theory*, vol. 47, pp. 1971–1988, 2004.

[13] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," in *Proc. 11th ACM Symp. Theory of Computing*, ser. STOC '79. ACM, 1979, pp. 1–12.

[14] H. L. Bodlaender and B. de Fluiter, *Parallel algorithms for series parallel graphs*. Springer, 1996, pp. 277–289.

[15] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 14, no. 5, pp. 1253–1257, 1993.

[16] S. Toueg and O. Babaoğlu, "On the optimum checkpoint selection problem," *SIAM J. Comput.*, vol. 13, no. 3, 1984.

[17] Pegasus, "Pegasus workflow generator." https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, 2014.

[18] T. Hérault and Y. Robert, Eds., *Fault-Tolerance Techniques for High-Performance Computing*, ser. Computer Communications and Networks. Springer Verlag, 2015.

[19] R. H. Möhring, "Scheduling under uncertainty: Bounding the makespan distribution," in *Computational Discrete Mathematics: Advanced Lectures*, H. Alt, Ed. Springer, 2001, pp. 79–97.

[20] L. C. Canon and E. Jeannot, "Correlation-aware heuristics for evaluating the distribution of the longest path length of a DAG with random weights," *IEEE Trans. Parallel Distributed Systems*, 2016.

[21] D. Sculli, "The completion time of PERT networks," *The Journal of the Operational Research Society*, vol. 34, no. 2, pp. 155–158, 1983.

[22] H. Casanova, J. Herrmann, and Y. Robert, "Computing the expected makespan of task graphs in the presence of silent errors," in *P2S2'2016, the 9th Int. Workshop on Programming Models and Systems Software for High-End Computing*. IEEE Press, 2016.

[23] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[24] R. M. van Slyke, "Monte carlo methods and the pert problem," *Operations Research*, vol. 11, no. 5, pp. 839–860, 1963.

[25] R. F. da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling research in distributed scientific workflows," in *e-Science (e-Science), 2014 IEEE 10th International Conference on*, vol. 1. IEEE, 2014, pp. 177–184.

[26] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2008, pp. 1–10.

[27] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[28] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[29] L. Han, "Checkpointing Workflows for Fail-Stop Errors: Simulation Code," https://doi.org/10.6084/m9.figshare.5057650.v3, 2017.

[30] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien, "Checkpointing workflows for fail-stop errors," INRIA, Research Report 9068, May 2017, short version appears in the proceedings of the IEEE Cluster Conference, 2017.

[31] C. Cao, T. Herault, G. Bosilca, and J. Dongarra, "Design for a soft error resilient dynamic task-based runtime," in *IPDPS*. IEEE, 2015, pp. 765–774.

[32] H. Jin, X.-H. Sun, Z. Zheng, Z. Lan, and B. Xie, "Performance Under Failures of DAG-based Parallel Computing," in *CCGRID '09*. IEEE Computer Society, 2009.

[33] E. Kail, P. fchtpen, and M. Kozlovszky, "A novel adaptive checkpointing method based on information obtained from workflow structure," *Computer Science*, vol. 17, no. 3, 2016.

[34] M. Chtepen, F. H. A. Claeys, B. Dhoedt, F. D. Turck, P. Demeester, and P. A. Vanrolleghem, "Adaptive task checkpointing and replication: Toward efficient fault-tolerant grids," *IEEE Trans. Parallel Distributed Systems*, vol. 20, no. 2, pp. 180–190, 2009.

[35] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *SC '14*. IEEE Press, 2014, pp. 719–730.

[36] J. A. Moreno, O. S. Unsal, J. Labarta, and A. Cristal, "Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart," in *23rd Euromicro PDP*, 2015, pp. 99–102.

[37] O. Subasi, O. S. Ünsal, J. Labarta, G. Yalcin, and A. Cristal, "Crc-based memory reliability for task-parallel HPC applications," in *IPDPS*, 2016, pp. 1101–1112.

[38] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 33, no. 6, pp. 518–528, 1984.

[39] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *J. Parallel Distrib. Comput.*, vol. 69, no. 4, pp. 410–416, 2009.

[40] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *ICS*. ACM, 2012.

[41] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello, "Lightweight silent data corruption detection based on runtime data analysis for HPC applications," in *HPDC*. ACM, 2015.

[42] L. Bautista Gomez and F. Cappello, "Detecting silent data corruption through data dynamic monitoring for scientific applications," *SIGPLAN Notices*, vol. 49, no. 8, pp. 381–382, 2014.

[43] ——, "Detecting and correcting data corruption in stencil applications through multivariate interpolation," in *FTS*. IEEE, 2015.

[44] K. Hashimoto, T. Tsuchiya, and T. Kikuno, "Fault-secure scheduling of arbitrary task graphs to multiprocessor systems," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, 2000, pp. 203–212.

[45] A. Girault and H. Kalla, "A novel bicriteria scheduling heuristics providing a guaranteed global system failure rate," *IEEE Trans. Dependable and Secure Computing*, vol. 6, no. 4, pp. 241–254, 2009.

[46] O. Subasi, G. Yalcin, F. Zyulkyarov, O. Unsal, and J. Labarta, "Designing and modelling selective replication for fault-tolerant hpc applications," in *CCGRID*. IEEE, 2017.

[47] A. Benoit, A. Cavelan, Y. Robert, and H. Sun, "Assessing general-purpose algorithms to cope with fail-stop and silent errors," *ACM Trans. Parallel Computing*, vol. 3, no. 2, 2016.

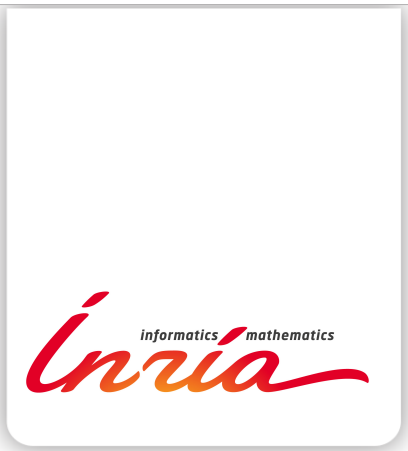
[48] G. Aupy, A. Benoit, H. Casanova, and Y. Robert, "Scheduling computational workflows on failure-prone platforms," *Int. J. of Networking and Computing*, vol. 6, no. 1, pp. 2–26, 2016.

[49] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 562–573.

[50] I. Assayad, A. Girault, and H. Kalla, "A bi-criteria scheduling heuristic for distributed embedded systems under reliability and real-time constraints," in *Dependable Systems Networks (DSN)*. IEEE, 2004.

[51] G. Jacques-Silva, Z. Kalbarczyk, B. Gedik, H. Andrade, K. L. Wu, and R. K. Iyer, "Modeling stream processing applications for dependability evaluation," in *Dependable Systems Networks (DSN)*. IEEE, 2011.

[52] J. Díaz, J. Petit, and M. Serna, "A survey of graph layout problems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 313–356, 2002.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399